图文 028、第4周答疑：本周问题统一答疑

1372 人次阅读 2019-07-28 07:00:00

详情 评论

第4周答疑：本周问题统一答疑

学员评论：

看了看线上一个服务，24天，FGC 121次，真恐怖。参数使用的默认的，没有进行设置。新生代的最大容量为66m左右,老年代133m，YGC 79000多次,FGC 121次。好可怕。

答：对的，你们的系统应该用户不太介意性能问题，不然这个系统早就不行了

问题

分配担保这里捋了一下有点疑问，如果没有设置分配担保失败的话，那么老年代可用内存小于新生代所有对象内存大小时直接full gc吗?

那这样如果新生代很大，老年代还剩余很大的内存就进行fullgc，是不是有点过于着急？感觉太浪费了。这个【允许分配担保失败】jvm默认是不允许的吗?

答：没错，没有老年代担保失败的设置，就会频繁触发Full GC，所以一般都要打开

问题

老师，请教一下，一个机器能开多少线程？取决于什么?

答：给你一个大致的数字：如果是4核CPU，本身JVM就有一些后台线程，还有你使用的一些框架可能也会有后台线程。

所以你自己的系统一般开启线程数量在几十个，比如50左右的一个数量，基本就差不多了。

大概系统所有线程加起来有100+，此时高峰期这么多线程同时工作，CPU负载基本就满负荷了

学员总结

个人感觉单线程和多线程各有优缺点，单线程是占用cpu资源少，对于单核的服务器来讲是不错的

而多线程对于多核cpu更好，与之带来的优势也很明显，就是多线程显然处理起来效率更高，使得stop the world 问题恢复的更快

问题

我想到一个问题：文章说的新老年代内存划分应该是在jvm中是一种类似“规范”的原理。

但是那个minor gc之前会检查老年代大小，然后后续等一系列判断。这些原理是所有版本的jvm都一致的吗。跟垃圾回收器的选择是否有关系？

比如如果选择了g1 作为收集器，是否在minor gc的过程也是一样？

答：基本上在g1之前都是这套原理，但是g1开始就变化了，从内存分配模型和垃圾回收机制，全部会变，所以后续有一周会专门剖析g1的原理

问题

老师，这个案例怎么也得16到17个线程同时并发处理，才能达到一分钟单台机器处理100个任务吧！

答：对的，生产系统单机开启了30+工作线程

问题

如果我在tomcat部署多个服务，JVM启动个数是根据服务来算的，还是根据tomcat来算的？我好像只能调试tomcat的堆栈分配情况。麻烦前辈解答下。

答：一个Tomcat启动多个Web应用，那JVM只有一个，就是Tomccat自己，你的Web应用都是一堆代码，由Tomcat根据 配置文件来对指定的请求调用你的代码

问题

看了今天文章，感觉和自己认知有点冲突，我之前一直的认知：minor gc是新生代的gc，major gc是老年代的gc，fullgc是新生代老年代和永久代的gc

marjor gc是包含在fullgc中的，然后stop the world会在full gc的时候发生,minor gc不会发生。这是我的认知。

但是文中说minor gc也会stop the world。这就有点冲突了。请老师指正，到底什么时候发生STW

答：看来你可能之前的认识是没有从本质认识垃圾回收和Stop the World

不管是老年代回收还是新生代回收，都要Stop the World，因为必须让程序别创建新对象，才能回收垃圾对象。

Full GC和Major GC其实是一个概念，都是指的老年代的GC，只不过一般会带着一次Minor GC，也就是Young GC，他们是一个概念多种名词

学员回答思考题

思考题，采用parnew+cms垃圾回收器如何只做ygc?

回答:和垃圾收集器没有什么关系，不同垃圾收集器，差别只在于性能和吞吐量的区别。并不影响垃圾回收时机。

根据堆中对象生存周期特点，合理分配eden s0 s1 大小，尽量让对象在新生代就被回收，需要注意的是要开启内存担保

答：是的，回答正确

问题

之前提到minor gc是很快的，这么多文章看下来没有特别明确说为什么快

我的理解是:新生代垃圾回收存活很少 且采用了复制算法，比标记整理效率高。 老师还有其他原因?

答：其实就这个原因，存活对象少，迁移内存很快，然后一次性清理垃圾对象，这个速度就是快。

老年代要先挪动对象压在一起，存活对象那么多，这里涉及到漫长的对象移动的过程，所以速度慢

问题

思考题： 一个面试题，parnew+cms的gc，如何保证只做ygc，jvm参数如何配置?

学员回答：我觉的应该考虑两个方向：避免FullGC和避免年轻代对象进入老年代

避免FullGC

1、保证老年代可用空间大于新生代所有对象，避免MinorGC前进行FullGC

2、如果1可以保证，那后面-XX：HandlePromotionFailure、进入老年代的对象平均大小等比较就不需要考虑了

3、保证MinorGC后存活对象不大于Survivor空间

避免年轻代对象进入老年代

1、根据实际情况查看每次MinorGC后存活对象的大小，设置合适的Survivor区域大小，保证存活对象进入survivor区，而不是进入老年代

2、根据对象存活的时间以及MinorGC的间隔时间，确定年龄。比如：3分钟一次MinorGC，而对象可以存活1个小时，那就把对象年龄设置到20，避免对象15岁进入老年代

3、大对象如果偶尔创建一个，可以设置-XX:PretenureSizeThreshold，使其分配至年轻代。如果创建销毁频繁，就让其直接进入老年代，利用对象池避免频繁创建销毁

答：你思考的非常好，其实在我们的那个案例里就说明了思路

问题

为什么新生代用的是复制算法?老年代用的是标记整理算法？既然复制算法比较快，为什么老年代不采用新生代的这种优化版的复制算法呢?

答：因为老年代的存活对象太多了，采用复制算法来回挪动大量的对象，效率更差

问题

我觉得老年代垃圾回收慢，是在并发标记阶段对所有老年代对象进行GC Roots追踪慢，要追踪到根源,而新生代的对象可能90%以上都不会存活,所以新生代gc快

补充，cms回收过程有4个阶段，会进入两次STW,这个也是影响速度的很大原因吧

答：是的，老年代垃圾回收，这个GC Roots追踪所有对象，因为老年代存活对象太多，所以耗时更长。还有考虑一下垃圾回收和整理这个环节，明天会分析

问题

这种每周总结一次答疑非常好，这样我们就不用每篇文章都去看评论了。老师已经把精华的问题总结出来了 ，非常好 点赞，爱你 么么哒

答：是的，每周的作业是一定要做的，认真做每周的作业，绝对可以把一周的内容吃透，消化成自己肚子里的东西，看答疑可以拓宽思路，看别人怎么提问的

问题

老师有一个问题，都已经full gc了，程序还并行运行，创建出来的对象放那？会一直触发full gc吗？

如果对象太多堆放不下，会等着full gc完成吗？这个时候也是世界停止吗？

答：对的，会继续放老年代，还可能会同步触发Minor GC，也可能有新的对象进入老年代，还可能有些老年代的对象失去了引用，啥都会发生，所以并发标记环节，很多是不准确的

为什么老年代垃圾回收比年轻代慢很多？学员自己的回答：

年轻代一般存活对象少，采用复制算法，从GC root出发标记存活对象，直接把存活对象复制到另一块内存，其余直接清除。

对于老年代，对象存活量大，每次遍历堆分别去标记存活对象和垃圾对象，再遍历把垃圾对象清除了，最后还要移动存活对象，防止太多内存碎片。

因为存活量大，耗时的地方我觉得在gc root引用的追踪还有存活对象的移动

答：对的，理解很好

问题

老师今天讲的cms垃圾回收器 初始标记 并发标记 重新标记 并发清理

其中重新标记老师说只会对并发标记改动的对象进行标记，是由什么结构存储了并发标记改动的对象吗？ 不然它这么快找到并发标记改动的对象

答：对的，他会内置记录在并发标记期间，被新建的对象，被变动的对象，有数据结构记录，所以这个阶段很快

问题

老师，当发生GC的时候，做标记的阶段，回收器是从Gc root出发去搜索吗？还是遍历堆里的所有对象，如果这样的话，该对象自身怎么知道谁在引用它呢？

还有第二阶段，它是怎么做到一边标记存活对象一边标记垃圾对象的？

答：

1、从GC Roots出发，去标记所有对象

2、他只要让垃圾回收线程工作的同时，让系统的工作线程也同时工作不就可以了

为啥老年代垃圾回收慢很多？学员自己的回答：

老年代的垃圾回收慢主要是因为剩余的存活对象很多，可能达到90%，这样子就不得不采用标记整理法

标记的过程中，并发标记是很慢的，因为对象变量要不断往上追踪看有没有gc root引用，不像方法局部变量和类变量那样直接被gc root引用，查找对象gc roots引用可能要向上追踪的次数比较多，所以耗时间

另外一个耗时间的地方就是并发清理，这个过程慢的原因是为了使得内存紧凑些，尽量不要出现内存碎片，是个边清理变移动的过程。

以上是我的分析，老师看我理解的对吗？

答：对的，理解的非常好

问题

eden区内存大小超过old，如果没有开启允许担保失败参数的话，岂不是young gc之前都会full gc了？

答：对的，所以开启那个担保机制

问题

通过GC Roots查找，直接或间接引用到的对象，就是存活的，进行标记。剩下的就是垃圾对象。在并发清除阶段就会清除了。 是这样吗？

答：是的

问题

针对web服务，POJO类一般都是在新生代，而通过@service @controller @component 等注解创建的对象一般都是在老年代。

针对一些纯java代码的后台跑批服务，基本都是新生代，除了一些通过静态变量或者常量引用的类，或者通过单例创建的类（本质也是通过静态变量引用）。

答：是的，总结的很好

问题

初始标记从GC Roots开始查找直接引用的对象。 并发标记是从对象出发，查看对象是否直接或间接有GC Roots引用由于并发标记会查看所有对象，且大多数对象都是存活的，所以过程会很耗时。 是这样吗?

答：其实并发标记也是从GC Roots出发的，通过每个对象的引用地址查看哪些对象是存活的，确实因为存活对象一般较多，所以很耗时

问题

full gc同时一般会伴随着一次minor GC，如果第一次full GC过程中，因为新创建的对象原因又达到了触发fullGC的条件，首先还是会先minorGC，然后尝试放入新生代

但此时老年代还没有回收完成，再触发一次MajorGC没什么意义，因为重新标记会干了这个事情。

如果minorGC后，新创建的对象仍然放不进内存，需要等待MajorGC结束，如果MajorGC结束后仍然放不进去，就会OOM了

答：总结的很好

问题

老师 有点没明白1 eden + 2 survivor的设计 为了优化空间利用 一定得是三块吗？为什么不能是两块？9:1不是也行吗

答：你看，如果一个eden一个survivor，那么回收的时候把两块区域的存活对象标记出来，放哪儿去？必须有另外一个survivor来存放，然后一次性回收之前的那个eden和survivor

问题

问一哈，之前课程新生代复制算法，老年代采用标记整理算法，为何CMS作为老年代垃圾回收，是采用标记清除算法呀

答：仔细看看今天的文章就明白了，CMS其实在标记-清理之后，会加入一个整理的过程，他是两个算法都用了

问题

新生代只需要一次"stop the world"的时间，在此期间完成标记清除并把存活对象转到survivor或老年代吗？

答：没错

问题

老师，有个问题要问您，因为CMS在“初始标记阶段”只标记直接引用。那么在“并发标记阶段”是不是是在初始标记阶段的基础上进行的

比如初始标记 标记了一个直接引用为A类，那么在并发标记阶段就不用再从A类的GC Roots开始了，直接从A类继续往下找就可以了。然后没有被遍历过的GC Roots接着被遍历就好了。

还有就是老师 CMS是标记-清除算法的话，那要是有个大对象进入到老年代没因为内存碎片问题放不下，那还得进行一次Full GC，如果GC后还是放不下，是不是就OOM了？

答：对的，可以节约时间。

明天的文章会讲解这个情况，这种对象是浮动垃圾，如果在cms gc期间进入老年代发现没有内存了，会引发concurrent mode failure问题，然后直接用serial old垃圾回收器进行回收，如果回收过后还会没地方放，那么才会oom

老年代的垃圾回收之所以比新生代慢很多，我觉得有以下几个原因：

1. ygc的把存活的对象直接复制到s区，而e区直接清空，不存在内存碎片化问题。

2. 从老年代的CMS而言，虽然它做到了尽可能的优化，但是其存在繁复的4步才能实现清理。

3. 本文介绍了老年代的垃圾回收机制，但是通过前面的文章我们知道，做了清理之后，是存在碎片化的问题，故而还需要搬运工把这些七零八落的存活对象重新排列，紧密的靠在一起，而老年代中存活的对象可能还是比较大的，那么就需要更多的搬运时间，这个过程也是比较耗时的。

另外还想请教老师一个问题：当解决碎片化问题后，那么这些对象的引用地址都会发生变化，那么该引用该对象的变量是如何修订引用地址的呢？

答：这个修改引用地址是全透明的，其实内存地址变了，引用的地址也修改就可以了

学员总结

判断对象是否存活通过遍历GC Roots，遍历过程中可能会发现，能被GC Roots 关联到的对象中，新生代可能占到了1% ，老年代占到了99%，然后从被关联到的对象出发追踪存活对象。

所以1%对象的追踪时间和99%的对象追踪时间是不一样的，也就是说老年代追踪时间大概是新生代的99倍。再加上老年代需要碎片整理。所以老年代垃圾回收时间比新生代长的多

问题

老师，会存在上一次垃圾回收还没执行完，然后垃圾回收又被触发的情况？

若有，jvm是怎么处理的，若没有，jvm是如何保证的呢？

答：新生代垃圾回收会直接Stop the World，系统不能运行了，所以必须等垃圾回收完了，才能再次gc；

老年代垃圾回收是有可能的，因为采用了并发收集的机制，一边回收，系统一边运行，也许没回收完就再次触发full gc，此时会进入stop the world，用serial old来回收

问题

看了昨天的评论，老师说了MinorGC其实也会Stop the World。我理解应该就是在跟踪复制阶段是不是？

如果真要用Stop，那么用时间来计算，是不是可以忽略不计，一般情况下。比如10ms，或者5ms就能完成这个步骤？烦请老师解惑下。

答：对的，新生代要stop the world，但是他速度很快，就是从gc roots出发标记出来少量存活对象，转移到空的survivor里，然后直接清空所有垃圾对象

问题

老师，我有个疑问，新生代我估算出来后，老年代该怎么设置。如果老年代过小了，也会有问题的吧。老年代和新生代有没有比较合适的分配比例。

答：没有通用比例，其实一切说通用的jvm优化参数，全都是瞎说的

因为通过目前20多篇文章的学习，大家就应该知道了，jvm参数优化，全部得基于每个不同系统的运行模型来分析，每个系统的jvm参数都应该是不同的，从看问题的本质，而不是找一个什么通用的比例

问题

请问在初始标记的时候是从GC Root出发找对象还是从老年代的每个对象开始看有没有被GC Root引用, 在并发标记的时候是从GC Root 出发还是第一次被标记过的对象开始, 在标记的过程中采用的是深度优先算法还是广度优先算法?

答：初始标记从GC Roots出发，仅仅标记直接引用的对象，但是并发标记也是从GC Roots追踪，只不过要一直顺着引用链条追踪看哪些对象是存活的

问题

我想请问一个问题就是通过什么方式可以得知系统进行了MinirGC或者FullGC?

答：有gc日志，还有很多的工具可以干这些事儿，后续我们会介绍的

学员评论

打卡，昨天又漏打卡了。跟着老师走，讲的言简意赅，比看深入理解虚拟机容易懂，配图好理解。看后自己绘图加深理解，基本JVM就容易回答面试题了。

结合实际项目考虑这些因素进去，事倍功半。老师辛苦了。

答：是的，接下来有两个案例解析，全程通过案例分析，教你们新生代和老年代的gc相关jvm参数如何根据系统去优化设置

问题

老师好，如果在垃圾回收时程序停止运行，那垃圾回收器用多线程回收还有什么意义呢，一样还是出现卡顿的现象

答：垃圾回收用多线程可以加快垃圾回收的速度啊，进而减少Stop the World的时间

问题

请问一下在Eden和Survivor区GC时也是从GC Root开始标记和跟踪对象, 在新生代的对象数量更多为什么在新生代就不耗时而在老年代GC的时候第二步并发清理的GC Root跟踪就很耗时?

答对的，GC Roots开始追踪，但是新生代存活对象极少，很快就追踪完毕了 ，老年代存活对象太多，追踪很耗时

学员评价

看过《深入JVM》一书和一些面试资料，再看本系列，还是有很大收获。

主要是老师讲的很有思路条理，能让读的人深入理解。

例如三区的分区思想，为啥分区，分区比例。 然后复制算法在三区里面怎么用的等等。 不止是理解这个技术点，还领会到了思路。

答多谢支持，继续加油学习

问题

标记清理后进行整理，不就是标记整理算法吗？为什么说两个算法都使用了呢？ 难道是我理解错了标记整理算法？标记整理算法不会有清理的过程？

答标记清理是先标记再清理，cms是这种算法，然后事后再整理；标记整理是先标记然后整理，最后才清理

问题

新创建的对象，到底是往Eden区放，还是Suivivor区域放？

我的理解是新的对象只会放入Eden中，而Survivor区只是用于存放每次Minor GC后的对象而已，新创建的对象是不会往Suivivor存放的

答没错，只放eden，survivor只放存活对象

问题

老师刚才的讲解解答了困扰我很久的疑惑，谢谢老师，那就是说，如果我需要控制对象不进入到老年代，那就必须保证s区存活对象不能超过50%，不然根据累加来看，年龄>（占比50%中的最大年龄）的对象就会进入到老年代。

答对的，所以要尽可能让每次gc后存活对象在survivor区的50%以内

问题

年轻代标记存活对象，那老年代的CMS回收器标记存活对象还是标记垃圾对象，如果只标记存活对象 ，那并发清理阶段，会有新的对象晋升到老年代，这些对象CMS可是没有做标记存活的，那清理的时候不就把这些对象也一起清理了吗?

答对的，所以文章里也说了，清理的时候仅仅清理标记的垃圾对象，然后并发清理的时候，新进入老年代的对象就是浮动垃圾，不会清理，所以可能会触发concurrent mode failure

问题

CMS设置的的空间占比1.6是92%，这之后就一直没有变化吗

发生ConcurrentModeFailure，启用 SerialOld 后，是只有SerialOld在工作，还是SerialOld和CMS一起工作

只有“老年代可用空间大于历次新生代GC进入老年代的对象平均大小”这种情况会考虑老年代-XX:CMSInitiatingOccupancyFaction设置的占比吗，其他两种FullGC情况不需要考虑吗，还是说所有的FullGC都会考虑这个占比

答

后面通过工具会让大家看到这些参数的值

那肯定只有Serial Old了，因为并发回收，人家一直创建对象，搞的老年代都不下了，赶紧要stop the world，让系统程序别运行了，然后Serial Old单线程直接回收，完事儿了，空出来了空间再让系统继续运行

条件独立，触发任何一个都会有Full GC

问题

老师，如果是通过动态年龄进入到老年代的对象，这一批对象的年龄最多也就是1吧，不可能熬了几轮才根据动态年龄进入到老年代（这一批对象第一次进入到s区时是1，第二次minorgc发现对象占用内存超过S区一半，直接进入老年代），麻烦老师对这个详细解惑下。

答对的，其实是这个年龄以及低于这个年龄的对象占据超过Survivor 50%，比如1岁，2岁，3岁的对象加起来占据了50%了，此时超过3岁的就会直接进老年代

问题

Handlerpromotionfailure参数是jdk8默认开启的吗。看完文章，我忍不住想调优一下了

答对的，默认开启的，可以的，就是要鼓励你们按照文章思路去分析自己的系统运行的内存使用模型，然后去尝试合理分配内存大小

前期要学会的优化就是一定要减少对象进入老年代，避免进行Full GC，尽量保证就是Minor GC，做到这一点，基本JVM就没什么大问题

问题

总结一下自己看完的感悟，目前看到老师案例的调优思路主要是两点。

第一，让常驻对象尽快进入老年代，以免留在新生代占用空间。

第二，避免让使用一次或者两次的对象(比如订单对象)进入老年代，从而可以避免漫长的full gc。

答是的，其实核心的JVM优化就是这个思路，前期把原理都学通了，悟透了，后续的案例会动手实操，结合更多案例来体验不同的场景优化

学员总结

感觉CMS就像一个手脚勤快的小伙子，在合适的时机进行并行工作，提高效率，但总有漏网之鱼（浮游垃圾）。

在小伙子顶不住的时候（没足够空间了，Concurrent Mode Failure），在背后注视着一切的老者（Serial Old）走了出来，大喊一声：Stop the world！有条不紊的把所有垃圾都清理掉，深藏功与名。

但是，但老者也顶不住的时候，就会发生灾难性的OOM。但据说，学了这门课的同学，都不曾见过这样的灾难。（如有不当之处，请老师指正）

答这个文采写的相当好，描述的很准确

学员总结

老师，我的一些总结和问题，请大家指正，谢谢！

对象进入触发 -->哪些情况下有对象进入老年代?

我知道的在下面

- 大对象直接分配到老年代 ---> 直接分配那如果一旦大于老年代可用也要触发fgc吧?
- minorgc的年龄到了15（有参数设置）
- 存活对象大于s区 --->这个存活对象包括年龄到了的对象吗?
- 动态年龄挤出对象 规则触发
    - Minor Gc前没有配置空间担保参数
    - Minor Gc前有配置空间担保参数，但老年代可用内存小于历次平均分配内存
    - minor Gc回收后的存活对象大于S区，大于老年代可用内存
    - 老年代已用空间达到CMSInitiatingOccupancyFaction 设置比例自动触发。

答：总结的非常好

学员总结

首先，计算系统高峰的qps，每秒内存开销=qps*单个对象大小*扩大20倍*其他操作10倍。 根据系统可用内存分配堆大小。分配年轻代大小根据每秒产生的内存开销来计算。

比如每秒60mb,2g给年轻代,默认eden:s1:s2为8:1:1,eden区有1.6g空间,26s左右会被占满，会进行minor gc。

此时存活的对象大概会有百分之10,160mb，这些存活的对象进入s1。有可能新生代回收的对象存活的可能在200mb以上， 那这样就会造成这些对象会直接进入老年代。这样就可以继续往上调新生代空间的大小,也可以调节eden区和s1 s2的比例。

空间担保参数也要打开，避免判断小于直接fullgc。系统中可能会有在内存中缓存大的对象,大的集合,这种对象一般都是要频繁使用或者要一直缓存的 ，这时候要设置直接晋升到老年代对象的大小。

还有在s1s2区设置晋升老年代年龄的设置，这个一般默认的我觉得目前就够用，要根据实际项目来设置. 垃圾回收器设置为parnew 还有cms。充分发挥多核处理器的优势。

整体来说，我觉得就是优化尽量避免频繁fullgc，降低系统STW的次数还有时间。

答非常好

问题

老师，有几个问题请教一下：

1、Minor GC前，年轻代中对象的总大小与老年代中可用内存比较，其中，可用内存是指剩余内存空间还是连续可用内存空间

2、文章中说，老年代会默认预留8%的空间给并发回收期间，进入老年代的对象使用，若进入老年代的对象大于8%的空间，是否会触发Full GC

3、若CMS在并发标记和并发回收时，进入老年代的对象又触发了Full GC，后一次的Full GC会立即执行还是等待前一次Full GC执行完毕

答

1、连续可用空间

2、其实之前说过，这种情况会触发concurrent mode failure，用serial old进行垃圾回收，直接stop the world

3、同上

问题

目前系统16G内存，jvm6G的内存，新生代5.5G，永久代设置128我M，老年代就是512减去128 等于384 M

minor gc大概450秒运行一次也就是8分钟 每次在eden区12M左右的对象 一个supervisor区 560M内存 每次回收之后大概占比空间的 百分之40到百分之60之间，supervisor空间是够的 第二占比 在百分之60一下 同年对象占比超过50的概率很低

之前晋升次数为31我 改成4了 一次minor gc8分钟，4次半小时，肯定需要进入老年代了

目前系统运行离更新 半个月 full gc 3次，所以我觉得堆内存就是合理优化minor gc 让对象不会那么快进入老年代 做出合理的预估 其实也不是堆越大越好 是需要根据系统整体运行情况预估 如果预估不准确 就用工具检测 然后合理优化

答：这个作业分析是我目前见过最好的一个，分析的非常好，看来你是彻底吃透我们目前讲解的内容了

问题

堆内存的调整 我觉得应该是观察 supervisor区 是不是 minor gc后占比过多 超过百分之70 可能就需要加大堆内存 或者说 业务高峰期 非常快就占满eden区 也需要加大堆内存 具体还有什么情况 请老师指正！

答

你理解没错，其实jvm优化的第一步，就是分析系统运行的内存使用模型，然后合理预估，合理分配内存，保证对象都在新生代里，进入老年代的速度要很慢很慢，做到这一点，jvm就是很完美的一个状态

问题

最后那个设置多少次FullGC之后进行碎片整理，我是否可以这样理解：

如果Full GC相对频繁一些，那就设置多次FullGC再碎片整理。

如果Full GC不是很频繁，可以设置每次FullGC都碎片整理，反正也占用不了多少时间。

这里先不考虑Full GC频繁是否调优，只是单纯以碎片整理这个参数怎么设置来考虑

答：是的，你说的很对，其实我觉得你基本都理解了